# Relatório Técnico

Núcleo de Computação Eletrônica

# Um Sistema Distribuído Para Algoritmos de Microeletrônica

Oliveira, C. E. T.
Pais, A. P. V.
Pereira, L. A.
Parga, D. F.
Anido, M. L.

NCE - 08/2000

Universidade Federal do Rio de Janeiro

# Um Sistema Distribuído Para Algoritmos de Microeletrônica

*Oliveira, C.E.T., Pais, A.P.V., Pereira, L.A., Parga, D.F. and Anido, M.L.*

Núcleo de Computação Eletrônica - Universidade Federal do Rio de Janeiro
E-mail: carlo@ nce.ufrj.br

## Abstract

*Microelectronics tools tend to consume large amounts of memory and processor time. When circuit size outgrows the resources available on a station, its time for a scalable tool architecture. Applied to design rule checking of flattened masks, this distributed, object-oriented architecture summons together the power of small, cheap desktop computers. The distributed system enables processing of larger circuits, assigning distinct parts of the problem to each machine. Larger circuits can be tested and testing time reduced as more computers are aggregated to the process.*

## Resumo

*Ferramentas de Microeletrônica tendem a consumir grandes quantidades de memória e tempo de processador. Quando o tamanho de circuito supera os recursos disponível em uma estação, surge a necessidade de uma arquitetura de ferramenta de escalável. Esta arquitetura distribuída orientada a objeto foi aplicada para conferir regras de máscaras planas, congregando o processamento de computadores de mesa pequenos baratos . O sistema distribuído habilita o processamento de circuitos maiores e aloca partes distintas do problema a cada máquina. Circuitos maiores podem ser testados agregando-se maior número de computadores que possibilitam maior uso de recursos e podem reduzir o tempo de processamento.*

Palavras chave: Orientação a Objetos, Sistemas Distribuídos, Sistemas *Multithread*, DCOM, Microeletrônica, Regras de Projeto

## Introdução

Ferramentas de Microeletrônica tem que estar preparadas para lidar com projetos que extrapolam os recursos de uma máquina. Uma maneira de criar ferramentas escaláveis é aplicar técnicas de objetos distribuídos, congregando um conjunto de máquinas para realizar a tarefa. Como exemplo o artigo implementa um verificador de regras de projeto acoplado a um editor de máscaras. Enquanto a edição se dá na máquina cliente, processos são distribuídos a outras máquinas para verificar a consistência da máscara editada. Esta distribuição é possível devido a estrutura encapsulada do modelo que suporta a distribuição em um sistema de três camadas.

## Arquitetura da Ferramenta

Esta ferramenta foi projetada para se integrar em um conjunto completo para produção de circuitos integrados. Ela possui uma estrutura modular [13] baseada no paradigma MVC. Diversos padrões de projeto foram aplicados para que fosse escalável e suportasse a inclusão de novos algoritmos agregados. O projeto usando a separação de canais de dados e controle visa a interoperabilidade entre diversos módulos. As caraterísticas mencionadas são a base para a implementação da arquitetura de forma distribuída.

Para demonstrar a modularidade da arquitetura, foi implementado o protótipo de um editor gráfico de *layout* de VLSI[5]. No protótipo implementado são apresentadas duas formas de visualização de um circuito integrado: uma gráfica (através de retângulos) e uma textual.

Ferramentas integradas de projeto para Microeletrônica são formadas pela agregação de módulos especializados e desenvolvidos por várias pessoas. A integração destes módulos necessita de um sistema flexível o suficiente para se adaptar às diversas especificações. Para isso é necessária uma camada de isolamento entre a interface e os diversos módulos.

A camada de isolamento é particionada em duas partes principais, uma para controle e outra para dados. O controle transfere comandos entre a interface e o módulo executivo. A ligação de dados provê uma correspondência entre a representação interna do módulo e a apresentação visual dos dados.

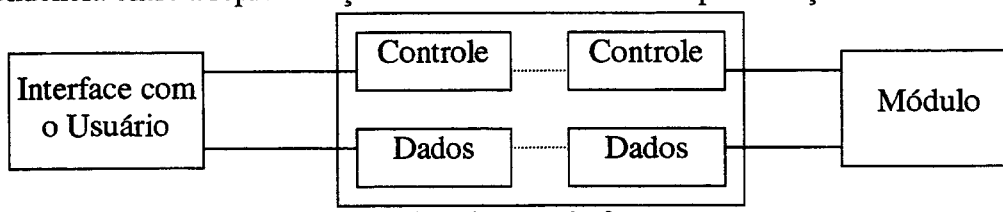



Figura 1 – Arquitetura da ferramenta

Para tornar esta arquitetura distribuída, foi necessário apenas reestruturar a camada de isolamento de forma que a comunicação entre a interface e os módulos fosse feita remotamente. Desta forma, tornou-se viável distribuir a execução de algoritmos por diversas máquinas.

## Modelo da Estrutura de Máscaras

A estrutura da ferramenta modela a descrição do circuito integrado a nível físico[5]. Uma máscara é descrita por um conjunto de retângulos em diversas camadas. Para se obter aceleração de algoritmos que lidam com retângulos, foi criada uma estrutura indexada a duas dimensões. A integração do

modelo com as outra partes do sistema é obtida com a aplicação de padrões de projeto que introduzem uma perturbação mínima na estrutura 2D. Estes mesmos padrões permitem que vários algoritmos sejam acoplados à estrutura sem ter que alterar seu código. Esta estrutura tem um encapsulamento que permite o seu transporte e distribuição em diversas máquinas.

A descrição de um CI pode se feita com CIF[4], uma linguagem que descreve retângulos. Esta linguagem estrutura a descrição do CI em células, camadas e caixas (retângulos). A cada comando CIF lido, é criado o objeto correspondente da estrutura. Uma célula corresponde a uma estrutura em árvore, onde a célula é a raiz, o primeiro nível é constituído pelas camadas, e o segundo nível, pelas caixas. Para cada comando de descrição de célula, camada ou caixa é criado o objeto respectivo, ou seja, é criado um *TCell*, *TLayer* ou *TBox*. Consequentemente, um *TCell* tem uma coleção de *TLayer*'s, e um *TLayer* tem uma coleção de *TBox*'s. Um arquivo CIF contém a descrição de uma ou mais células. Para representar diversas células, foi criado um objeto *TLayout* que contém cada objeto *TCell* criado. O objeto *TLayout* corresponde à descrição completa do arquivo CIF.

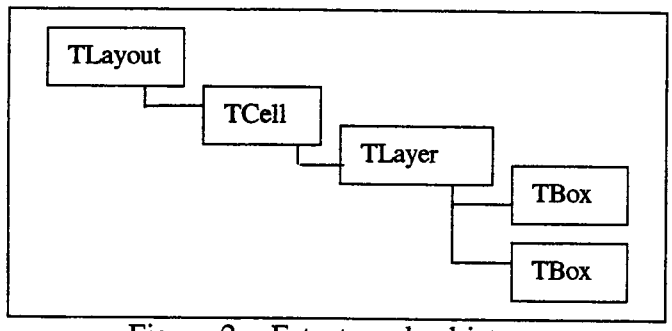



Figura 2 – Estrutura de objetos

O modelo incorpora um significado semântico que precisa ser tratado pela ferramenta. Este tratamento envolve a correlação entre retângulos numa mesma camada ou entre camadas. Como a descrição de um CI pode envolver milhões de retângulos, a manipulação de listas extensas de retângulos implica num custo muito alto de tempo e em baixa performance. Para se ter eficiência neste tratamento é preciso que esta estrutura seja indexada.

A estrutura indexada foi concebida a partir do conceito de *Span*[9]. Um *Span* é um intervalo nas coordenadas X ou Y. Para representar um retângulo é necessário um *Span* no eixo X e outro no eixo Y. Uma camada é descrita nesta estrutura por um Plano. Um Plano é uma coleção de *SpanY*'s. Um *SpanY* é formado por um intervalo no eixo Y e por uma coleção de *SpanX*'s. Um *SpanX* é apenas um intervalo no eixo X. O intervalo do *Span* é representado por um par Origem/Destino. As coleções de *Spans* são ordenadas pela Origem de cada *Span*. Uma restrição imposta é que não há *SpanX*'s consecutivos, mas *SpanY*'s podem ser consecutivos.

Anteriormente, *TLayer* foi definido como uma lista de *TBox*'s. Mas, para suportar a estrutura indexada *TLayer* é formado por um plano de *Span*'s. Quando um *TBox* é adicionado ao *TLayer*, são criados os *SpanX* e *SpanY* correspondentes. O *SpanX* é adicionado ao *SpanY*, e o *SpanY* é adicionado ao Plano que compõe o *TLayer*. Desta forma, não é necessário que o *TLayer* mantenha uma lista de *TBox*'s.

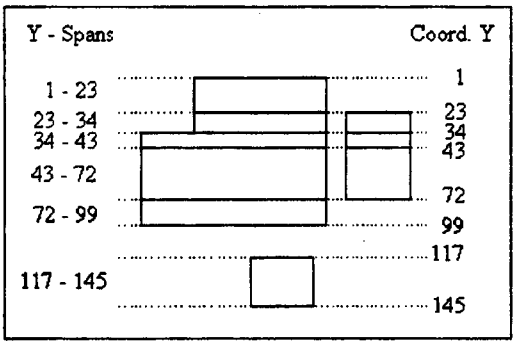



Figura 3 – Estrutura de *Spans*

A estrutura de *Spans* é projetada para acelerar a consulta de objetos relacionados. Esta consulta inclui operações de inclusão, intercessão, união, inflação, proximidade e exclusão. A aceleração é obtida tanto pela indexação 2D como por uma busca binária nas listas.
Vejamos, por exemplo, o algoritmo correspondente a consulta da camada resultante da inflação de uma camada. Usando a estrutura de *Spans*:

*Cria camada G.*
*Para cada SpanY da camada A,*
    *Cria o SpanY SY.*
    *SY = (Origem-1,Destino+1).*
    *Para cada SpanX do SpanY*
        *Cria o SpanX SX.*
        *SX = (Origem-1, Destino+1).*
        *Insere SX no SpanY SY.*
    *Insere SY na camada G.*
*Retorna a camada G.*

Se estivéssemos usando uma lista de retângulos, cada retângulo da lista seria inflado de modo similar. Com isso alguns retângulos poderiam ser englobados por outros. Portanto ainda seria necessário retirar os retângulos englobados, o que implicaria comparar cada retângulo da lista aos demais retângulos.
Para esta estrutura ser disponibilizada para outra partes do sistema foram aplicados os padrões de projeto[2] *visistor*[2], *iterator*[2] e *decorator*[2]. A grande vantagem destes padrões é disponibilizar o conteúdo de uma foram encapsulada, sem deixar transparecer seus detalhes. Além disso torna a estrutura flexível e adaptável à inclusão de novos algoritmos. Com isso o modelo fica bastante ortogonal sendo enxergado por diversas operações do sistema da mesma maneira.
Para realizar qualquer operação na estrutura, é necessário varrer listas de objetos. Para isso foi usado o padrão *iterator*[2]. A aplicação deste padrão consiste em criar um objeto responsável pela varredura do objeto composto, sem expor sua representação interna. Desta forma, para cada objeto composto da estrutura é criado um objeto TIterator. TIterator tem uma referência para o composto e expõe uma interface que permite que os elementos do composto sejam acessados seqüencialmente. Desta forma é o TIterator que mantém o estado atual da varredura, permitindo que sejam feitas várias varreduras no mesmo composto. Além disso, o composto fornece um método que retorna o seu *iterator* correspondente. Como exemplo há um trecho de código em *Delphi* que faz uma operação polimórfica num composto:

```
Iterator := Composto.Iterator;
Iterator.PrimeiroElemento;
while not Iterator.Terminou do
  Iterator.ProximoElemento.Operacao;
```

Figura 4- Padrão *Iterator*

Para fazer a operação de pintura da estrutura na tela, foi usado o padrão *visitor*[2]. O *visitor* representa uma operação a ser realizada em todos os elementos da estrutura. Para isso é criado um objeto TVisitor que executa a operação de pintura em cada elemento da estrutura. Com a flexibilidade obtida pelo uso do *visitor*, esta solução foi estendida para outras operações realizadas na estrutura.

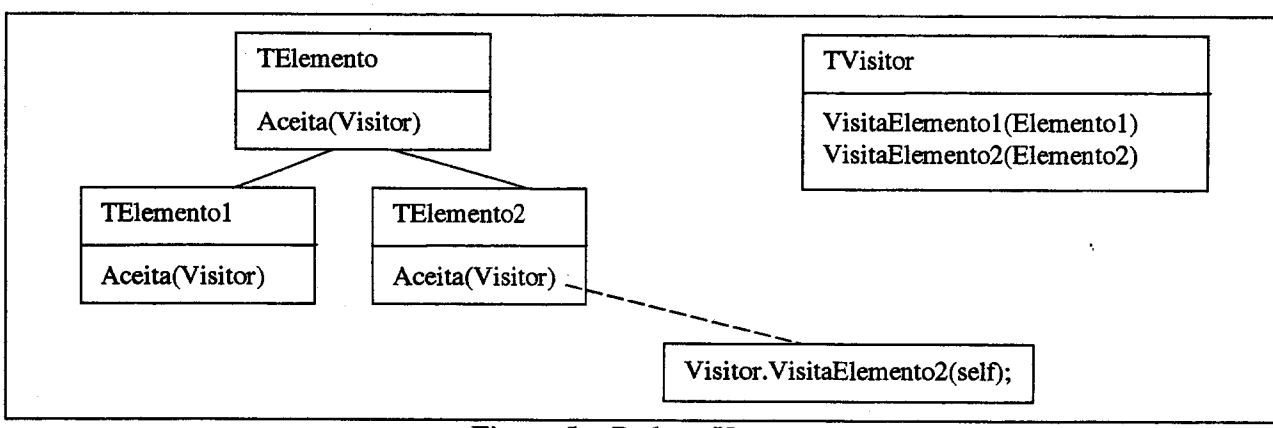
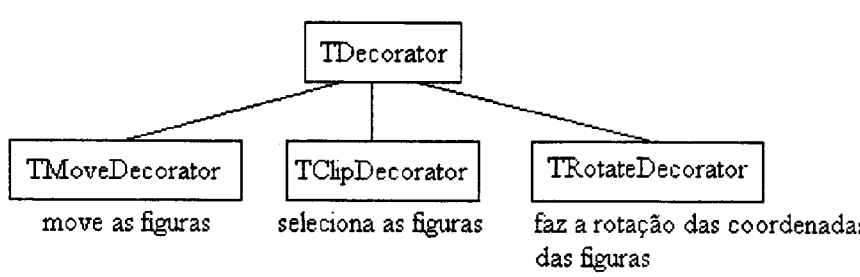



Figura 5 – Padrão *Visitor*

As operações realizadas na estrutura podem envolver transformações de coordenadas como espelhamento, rotação e transladação. Tomando como exemplo a operação de pintura, seria necessário criar objetos que fizessem a pintura com cada uma dessas transformações e também a combinação dessas transformações. Ao invés disso, foi usado o padrão *decorator*[2]. Este consiste numa alternativa flexível para o uso de subclasses, já que as responsabilidades são adicionadas ao objeto dinamicamente. Para isto, basta criar um objeto TDecorator que faz a operação básica. Neste caso, o TDecorator é o TVisitor. E para cada transformação é criado um derivado do TDecorator. No exemplo abaixo, foi criado um objeto com três funcionalidades que foram unidas dinamicamente.




```
Painter := TRotateDecorator.Ceate ( axe,
                TMoveDecorator.Create ( XOffset, YOffset,
                        TClipDecorator.Create ( Rect, TDecorator.Create ) ) );
```

Figura 6 – Padrão Decorator

A estrutura do modelo se presta prontamente à partição em objetos distribuídos. As células podem ser particionadas em suas camadas com estas sendo alocadas em diversas máquinas. Para transportar uma camada para outra máquina, é iniciada uma varredura de forma que cada *SpanY* é enviado por vez. Cada *SpanY* é codificado para uma seqüência de caracteres que é decodificada para o objeto *SpanY* na máquina destino. Para isso, foi usado o componente TPolymorphicList[13] que converte objetos para uma seqüência de bytes, que facilmente é convertida para caracteres.

A estruturação do modelo com indexação 2D prevê a adição de diversos algoritmos relacionados à microeletrônica. Como exemplo disto, um algoritmo que extrai a lista de transistores das máscaras foi implementado na mesma estrutura. A ordenação da estrutura em segmentos verticais e horizontais permite que a complexidade algorítmica caia de $N^2$ para O[logn][8]. Toda a estrutura foi projetada tendo em mente a integração com outros módulos e o encapsulamento e partição em objetos distribuídos. Ela está na base da arquitetura que suporta a escalabilidade da ferramenta permitindo o uso de algoritmos distribuídos.

## Arquitetura Distribuída

A criação de um sistema distribuído a partir de um outro já existente, normalmente envolve um custo de engenharia. É desejável que este custo seja mínimo. O desdobramento da arquitetura da ferramenta em um sistema distribuído é uma conseqüência da sua estrutura original MVC. O sistema é implementado em três camadas, mantendo a lógica de edição no cliente, o controle de distribuição de tarefas na camada central, enquanto o modelo executa o algoritmo distribuído. No controle central, uma *thread* controla um conjunto de filas que gerencia os recursos do sistema[11]. Estes recursos são representados por um objeto *proxy*[2] que mapeia o objeto remoto. O objeto remoto implementa a geração de uma nova camada a partir de uma ou mais camadas existentes. Os objetos remotos são implementados como objetos DCOM[3], assim como os proxies e a camada central.

A proposta da arquitetura distribuída é feita segundo uma política de mínima intervenção na arquitetura original. O paradigma MVC já existente se desdobra na formação de um sistema de três camadas.

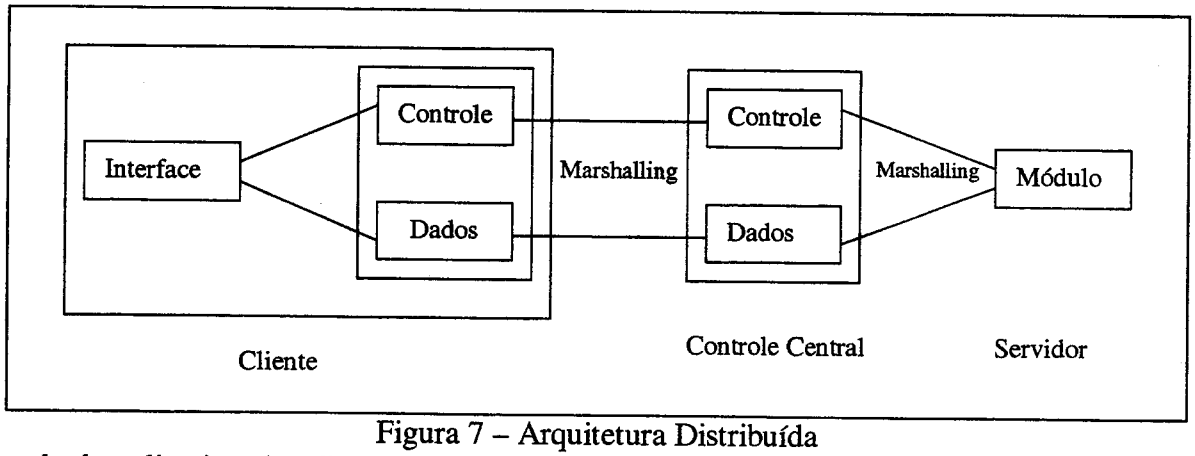



Figura 7 – Arquitetura Distribuída

O controle de aplicativo, localizado no cliente, envia comandos através do canal de controle. O canal de controle implementa o padrão *Chain Of Responsability*[2], que provê o encaminhamento de mensagens através dos controladores localizados nas camadas cliente e controle central. Os canais de dados entre o cliente e controle central, e entre o controle central e o servidor são estabelecidos através

do *marshalling* de objetos. A transmissão de dados é feita entre objetos remotos, que se comunicam pelo protocolo DCOM.

Na camada central reside o controle de distribuição. O processamento do algoritmo de verificação é controlado por um *script* em XML[14] que descreve um conjunto de regras. O *script* é processado por um *parser* que gera comandos a serem executados remotamente. Esses comandos provocam a criação de pseudocamadas, resultantes de operações entre camadas. O *builder*[2] classifica as pseudocamadas. Se a pseudocamada só depende de camadas originais, então a operação já pode ser executada. Caso contrário, ela terá que esperar por seus recursos. As regras de verificação são ordenadas manualmente no arquivo XML, isso porque escolhemos uma heurística simplificada. Poderíamos ter implementado um algoritmo que percorresse o grafo de dependência entre as regras e ordenasse as regras de acordo a precedência destas.

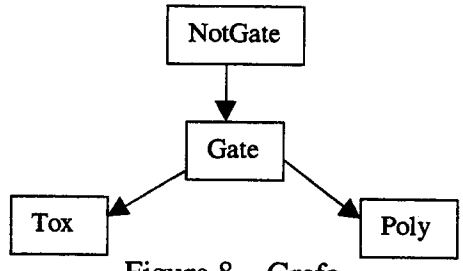


Figura 8 – Grafo

A figura acima mostra como deveria ser a estrutura do grafo. NotGate é uma pseudocamada que depende de uma outra pseudocamada a Gate. Por sua vez, *Gate* depende de suas camadas originais *Tox* e *Poly*. Estes pseudocamada são objetos *proxy*[2], que constituem os consumidores e produtores do sistema.

Utilizamos o padrão *command*[2] para fazer um refinamento na solução do problema. Existem diversas regras e dependendo do tipo da regra, uma seqüência diferente de comandos é executada. O parser dispara a execução do command. O *command* ,por sua vez, envia requisição para o builder, como ilustrado na figura abaixo.

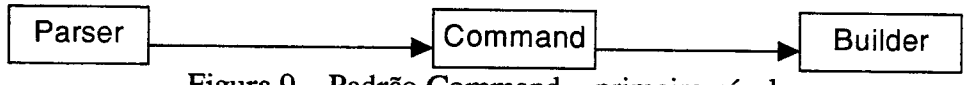


Figura 9 – Padrão Command – primeiro nível

O padrão command também atua num nível posterior. Neste caso, o proxy assume o papel do command. Dependendo do tipo de proxy que está sendo executado, um bloco diferente de comandos é chamado. Assim, a estrutura do padrão é recursivamente utilizada. A figura abaixo ilustra esse nível. O controle central dispara uma thread de execução. A thread executa o proxy e este, por sua vez, executa o servidor.

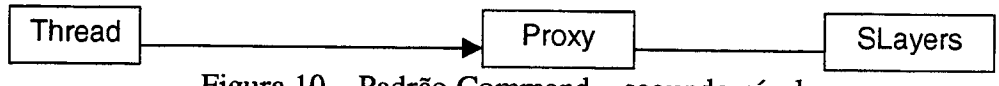


Figura 10 – Padrão Command – segundo nível

O controlador define uma *thread* que gerencia os recursos a partir de dois monitores[11]. Um dos monitores controla a liberação dos processadores. Este monitor é incrementado sempre que uma máquina termina sua execução, e decrementado sempre que uma máquina é alocada a uma tarefa. Inicialmente, este monitor é incrementado para cada máquina disponível. O outro monitor controla a fila de tarefas prontas. Ele é incrementado sempre que há uma nova tarefa a ser executada e decrementado quando a tarefa é alocada a uma máquina. Este monitor é necessário, para evitar que a tarefa seja retirada da fila antes que esteja pronta para ser executada.

Os proxies desempenham um papel primordial na gerência de recursos. O controle central gerencia os recursos a partir de três filas de objetos. A fila de espera contém os objetos que precisam de insumos. A fila de prontos contém os objetos que não precisam de insumos. Quando uma tarefa necessita do resultado de uma outra tarefa, o objeto correspondente é alocado a fila de espera. Mas se a tarefa necessita de uma camada primitiva, o objeto é alocado a fila de prontos. Além disso, é necessária uma fila de execução para as tarefas que estão em execução. Esta fila reflete os processadores que estão ocupados. Desta forma, quando a tarefa termina de ser executada, o objeto correspondente é removido da fila de execução, e o processador é alocado para outra operação.

Quando o controle central invoca o primeiro *proxy* da fila de prontos, este *proxy* inicia um processo servidor de recursos. Do ponto de vista do controle central, este *proxy* é um produtor de recursos, enquanto que para o servidor, ele é um consumidor de recursos. Para que o *proxy* possa desempenhar esses papéis, ele implementa o padrão *Observer*[2]. O consumidor monitora os produtores que são os insumos necessários a execução da tarefa que ele corresponde. O produtor mantém uma lista dos seus consumidores. Quando a tarefa acaba de ser executada, o produtor avisa o término da operação a todos os consumidores que o observam.

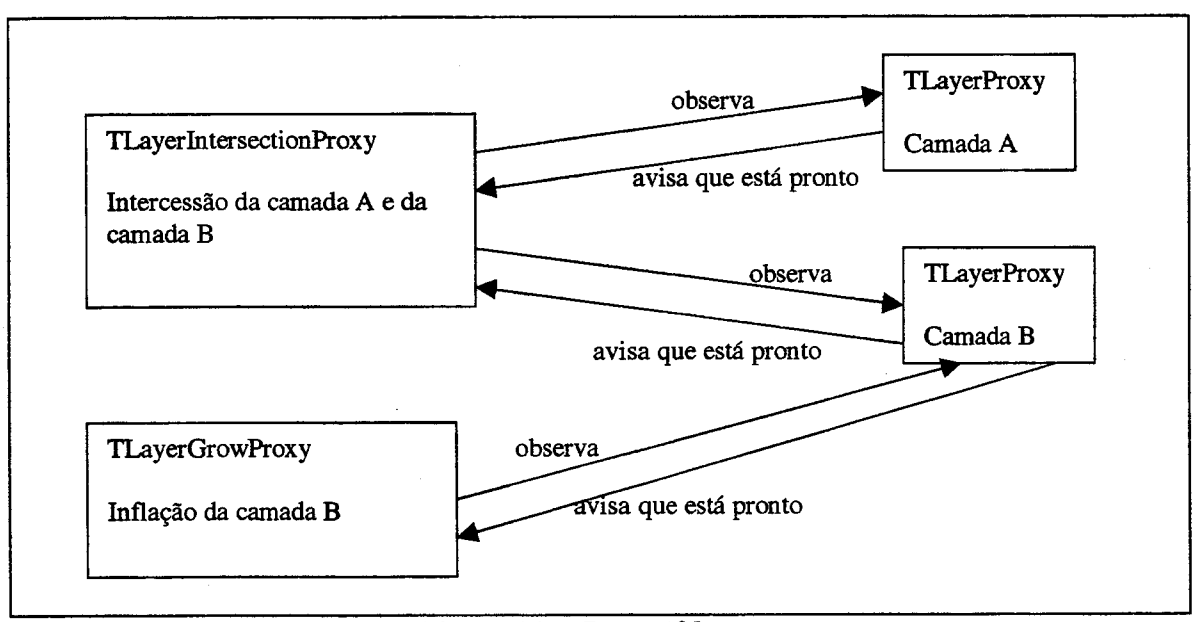



Figura 11 – *Proxy/Observer*

A figura acima ilustra a operação de intercessão de duas camadas, que tem como resultado uma nova camada. A operação só pode ser executada quando as camadas A e B estão prontas. Assim, quando o

proxy da intercessão é notificado pelas camadas, ele executa a operação de intercessão e notifica os seus observadores que ele também está pronto. Além disso, ele notifica o controlador do término da operação e é promovido para a fila de prontos.

A arquitetura descrita implementa um sistema produtor/consumidor[11] distribuído. A sua concepção foi baseada em técnicas de orientação a objetos. Estas técnicas promoveram o encapsulamento das funcionalidades dos produtores e consumidores, e a reutilização de código nas diversas camadas. Além disso, esta arquitetura é aplicável tanto a implementação de heurísticas simples, como a que foi usada, quanto a implementação de heurísticas complexas. A padronização obtida a partir da linguagem de descrição XML permite que a arquitetura seja flexível o suficiente para suportar a implementação de outros algoritmos.

## Execução do Algoritmo

O algoritmo foi testado usando um circuito muito simples que é modificado aleatoriamente para geração de erros. O programa utiliza dois arquivos: um para escrita e outro para leitura. O arquivo destinado a leitura contém os seguintes dados: um determinado numero de retângulos, que são os principais causadores dos erros; uma determinada área de circuito integrado, onde os erros são gerados. Já o arquivo de escrita guarda o resultado de todas as configurações.

Ao ser executado, o programa lê os dados do arquivo e insere retângulos de diversos tamanhos e posições em diversas camadas dentro da área de circuito integrado. O objetivo é testar o tempo de execução e a capacidade da memória dos computadores utilizados. Todos os resultados são gravados no arquivo destinado a escrita, podendo ser lido sempre que desejado. Este arquivo contém: o número de retângulos; a área de circuito integrado; a data em que aquela configuração foi realizada; a hora que o programa começa a ser executado (tempo inicial); a hora que o programa termina de ser executado (tempo final); a diferença entre os tempos final e inicial;
Diversas configurações de erros foram testadas. A bateria de testes utilizou computadores IBM PC Pentium 166Mhz , com 64 Mb de memória, ligados em uma rede de 10Mbits. Os resultados de algumas destas configurações são citadas na tabela abaixo.

| Número de Retângulos | Área de Circuito Integrado | Tempo de Execução do Programa (h:m:s:ms) | Numero de Maquinas Utilizadas |
|---|---|---|---|
| 800 | 800 | 0:7:10:419 | 3 |
| | | 0:7:05:219 | 5 |
| 800 | 400 | 0:6:06:988 | 5 |
| | | 0:5:59:297 | 3 |
| 1000 | 400 | 0:5:13:811 | 4 |
| | | 0:2:36:855 | 1 |

Tabela 1-Execução do algoritmo distribuído

Verificamos na tabela acima que, mantendo o numero de retângulos, o tempo de execução e inversamente proporcional a área de circuito integrado. Basta compararmos a primeira com a quarta linha. Podemos ver que o tempo de execução sofre alterações significativas. Isso se deve a uma maior

densidade de retângulos. A inserção de um retângulo numa área onde já exista spans, acarreta em aumento de processamento.

O resultados confirmam a escalabilidade, uma vez que muitas das operações de alta densidade só puderam ser completadas com um número grande de máquinas. No entanto os resultados foram contrários ao esperado em termos de performance. A heurística simplificada de alocação de tarefas é uma das responsáveis pela degradação do tempo de resposta. Como não foi observado o grafo de dependências, são provocados um número excessivo de transferências entre máquina. O próprio processo de transferência é feito através de um *marshalhing* não otimizado. Também nesta implementação simplificada os servidores executavam uma única *thread*, ficando ociosos durante o transporte de dados.

## Conclusões

Arquiteturas escaláveis distribuídas são uma solução econômica para o problema do crescimento constante dos sistemas em geral e em particular para ferramentas de CAD. Elas apresentam uma solução melhor do que alocar em uma máquina uma grande quantidade de recursos. Estes recursos podem ser melhor aproveitados se distribuídos entre diversas estações de trabalho.
O projeto original do sistema apresentado foi cuidadosamente pensado para ser adaptável para um modelo escalável distribuído. A modificação necessária se resumiu a acrescentar um controlador de recursos distribuídos. A estrutura de objetos comportou facilmente o modelo distribuído e foi aproveitada sem mexer nenhuma linha de código. Com isso se obteve uma plataforma básica escalável onde podem ser desenvolvidos diversos algoritmos distribuídos. Quando se precisa aumentar a capacidade computacional basta agregar mais máquinas na lista de hospedeiros.

Apesar dos resultados contrários na questão de performance, o experimento demonstrou a viabilidade de se implementar um sistema escalável. Este trabalho servirá como plataforma para desenvolver novas heurísticas e técnicas de alocação e transferência de recursos entre máquinas, balanceamento e realocação de carga. Outros algoritmos já implementados na versão concentrada da ferramenta vão ser migrados para o modelo distribuído. Algoritmos de extração, simulação, roteamento e alocação poderão validar a flexibilidade da arquitetura e a aplicabilidade do modelo distribuído para escalabilidade de recursos de processamento.

## Referência Bibliográfica


[1] Furlan, J.D., "Modelagem de Objetos através da UML – The Unified Modeling Language", MAKRON Books, 1998.

[2] Gamma, E., Helm, R., Johnson, R., Vlissides, J., "Design Patterns : Elements of Reusable Object - Oriented Software", Addison-Wesley, 1998.

[3] Eddon, G., Eddon, H., "Inside Distributed COM" – Microsoft Press, 1998.

[4] Weste, N., Eshraghian, K., "Principles of CMOS VLSI Design", Addison-Wesley,1988.

[5] Mead, C., Conway, L., "Introduction to VLSI Systems", Addison-Wesley,1980.

[6] Oliveira, C.E.T. e Anido, M.L., "TEDMOS para Windows", IX Congresso da Sociedade Brasileira de Microeletrônica, Campinas, pp. 65-73, Agosto, 1994.

[7] Nunes, R.B., Anido, M.L. e Oliveira, C.E.T., "Circuit Verification Using Spans – A DataStructure with O(n) Algorithms", IX Congresso da Sociedade Brasileira de Microeletrônica, Campinas, pp. 65-73, Agosto, 1994.

[8] Nunes, R.B., Anido, M.L. e Oliveira, C.E.T., "A New Approach to Perform Circuit Verification Using O(n) Algorithms" , IEEE Procedings of the EUROMICRO'94 conference, Liverpool, IEEE Computer Society Press, pp. 428-434, 1994.

[9] Alcântara, J.M.S., Oliveira, C.E.T. e Anido, M.L., "A Novel Circuit Extration Tool Based on X-Spans and Y-Spans", IEEE Procedings of the 21st EUROMICRO Conference, Prague, Tcheck Republic, Setembro, 1996.

[10] Nunes, R.B., Anido, M.L. e Oliveira, C.E.T., "A New Approach to Perform Circuit Verification Using Spans" , IEEE Procedings of the 38th Midwest Symposium on Circuits and Systems, Agosto, Rio de Janeiro, Brasil, 1995.

[11] Stallings,W., "Operating Systems – Internal and Design Principles" – 3ª edição- Prentice Hall,1997.

[12] C.E.T. Oliveira, A.L.C.L. Duboc, A.P.V. Pais, D.P. Muniz, M.L. Anildo. "Aplicações de Patterns no Desenvolvimento de Um Sistema CAD para Microeletrônica" Núcleo de Computação Eletrônica, UFRJ,1999.

[13] Web - http://www.tecepe.com.br/omar.

[14] Wrox Development Tem, Ducket, J. "Professional XML"-2ª edição - Wrox Press